2-2020

Automated deprecated-API usage update for Android apps: How far are we?

Ferdian THUNG
Singapore Management University, ferdianthung@smu.edu.sg

Stefanus AGUS HARYONO
Singapore Management University, stefanusah@smu.edu.sg

Lucas SERRANO
Sorbonne University

Gilles MULLER
Sorbonne University

Julia LAWALL
Sorbonne University

See next page for additional authors

Citation

THUNG, Ferdian; AGUS HARYONO, Stefanus; SERRANO, Lucas; MULLER, Gilles; LAWALL, Julia; LO, David; and JIANG, Lingxiao. Automated deprecated-API usage update for Android apps: How far are we?. (2020). *2020 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER): 18-21 February, Ontario, Canada: Proceedings*. 602-611.
Available at: https://ink.library.smu.edu.sg/sis_research/5502

Automated Deprecated-API Usage Update for Android Apps: How Far Are We?

Ferdian Thung*, Stefanus A. Haryono*, Lucas Serrano†, Gilles Muller‡, Julia Lawall‡, David Lo* and Lingxiao Jiang*
*School of Information Systems, Singapore Management University
Email: {ferdianthung,stefanusah,davidlo,lxjiang}@smu.edu.sg
†Sorbonne University/Inria/LIP6
Email: Lucas.Serrano@lip6.fr
‡Inria
Email: {Gilles.Muller,Julia.Lawall}@inria.fr

***Abstract*—As the Android API evolves, some API methods may be deprecated, to be eventually removed. App developers face the challenge of keeping their apps up-to-date, to ensure that the apps work in both older and newer Android versions. Currently, AppEvolve is the state-of-the-art approach to automate such updates, and it has been shown to be quite effective. Still, the number of experiments reported is moderate, involving only API usage updates in 41 usage locations. In this work, we replicate the evaluation of AppEvolve and assess whether its effectiveness is generalizable. Given the set of APIs on which AppEvolve has been evaluated, we test AppEvolve on other mobile apps that use the same APIs. Our experiments show that AppEvolve fails to generate applicable updates for 81% of our dataset, even though the relevant knowledge for correct API updates is available in the examples. We first categorize the limitations of AppEvolve that lead to these failures. We then propose a mitigation strategy that solves 86% of these failures by a simple refactoring of the app code to better resemble the code in the examples. The refactoring usually involves assigning the target API method invocation and the arguments of the target API method into variables. Indeed, we have also seen such transformations in the dataset distributed with the AppEvolve replication package, as compared to the original source code from which this dataset is derived. Based on these findings, we propose some promising future directions.**

***Index Terms*—API usage, program transformation, Android, mobile apps**

I. Introduction

In recent years, each release of Android has seen the deprecation of some APIs and the introduction of new ones. Due to the fragmentation of the Android market [12], [15], app developers must update their apps to use the new APIs, while maintaining backward compatibility. Unfortunately, updating API uses is not always trivial. While API changes are described in the documentation, these descriptions are not always accompanied by concrete examples. When an API is used at many places in the app source code, updating its uses is tedious and error prone.

AppEvolve was recently proposed by Fazzini et al. [10] to automate the updating of Android apps in response to deprecations of Android APIs. The main idea of AppEvolve is that many software projects may use the APIs in similar ways, and thus existing codebases may already contain examples of how to update uses of deprecated APIs. Given a target app that uses a deprecated API and information about the new API method(s) that should replace it, AppEvolve collects update examples from existing codebases, abstracts their variable references, ranks them according to their expected applicability, and then tries to apply the abstracted examples one by one to the target app. If an abstracted example matches the code, the update is said to be *applicable*. Applicable updates are furthermore validated by testing. A key feature of AppEvolve is that it does not merge the abstracted examples into a single generic pattern, to avoid losing information that may be specific to particular usage contexts.

AppEvolve has been evaluated on a dataset involving 15 real-world apps and 20 real API changes. The dataset contains 41 usage locations of these real API changes. AppEvolve successfully produced validated applicable updates for 85% of these API changes and 90% of their usage locations. The paper on AppEvolve is accompanied by a replication package, containing the tool and the dataset.[1] We have reproduced the results of AppEvolve on this dataset.

In this work, our goal is to understand the generalizability of the results reported for AppEvolve, and to analyze the characteristics of the apps for which AppEvolve cannot generate applicable updates. To this end, we consider the same APIs and change examples as used in the AppEvolve evaluation, but create a new dataset of target apps. In selecting the new target apps, we make no effort to take the expected strengths or weaknesses of AppEvolve into account, and only check that the apps use one of the deprecated APIs considered in the AppEvolve dataset and do not use the corresponding replacement API. Of the apps we found at GitHub that satisfy these properties, we randomly selected 54 mobile apps that are entirely distinct from the original data set, and that involve 14 of the 20 real API changes and 54 usage locations.

We applied AppEvolve to update the deprecated-API usages in these mobile apps and analyzed the cases where it fails to update the apps. Indeed, running AppEvolve on our dataset generates applicable updates for only 10 out of 54 usage locations. It generates only failed updates, i.e., updates that do not match the deprecated-API invocation, for the remaining 44 usage locations. Our analysis shows that AppEvolve produces

[1] https://sites.google.com/view/appevolve

applicable updates for a mobile app in cases where the API usage closely matches an API change example. On the other hand, AppEvolve often fails to update a mobile app when there are minor syntactical differences between the invocations of the deprecated API in the target app and the change example. It also fails when there are semantic differences, such as the use of inheritance or the use of `static` or `final` modifiers, when encountering edits that require modifications beyond method boundaries, and when finding no relevant change examples in GitHub. Of these limitations, only the latter two were mentioned in the AppEvolve paper.

To try to understand the issues leading to the drastic performance difference as compared to the results of the original AppEvolve evaluation, we refactor the use of the deprecated in the target app code so that it more closely resembles the uses found in the change examples from which AppEvolve learns the edits. After refactoring, we found that AppEvolve can generate applicable updates for 86% of the cases where it previously failed to do so. Based on these results, we propose to mitigate the dependence of AppEvolve on the code structure of the examples and the deprecated API usage by normalizing the code to a standard form coupled with the use of identifier-name recommendation, to improve existing work on automated deprecated-API usage updates for Android apps.

The main contributions of this paper are as follows:

- We evaluate AppEvolve with additional Android apps to investigate if it can effectively perform API-usage updates in different situations. The dataset that we created contains three times more real-world apps than those used to evaluate AppEvolve in the original paper (54 vs 15).
- We categorize the limitations of AppEvolve and show API usage instances that are affected by these limitations.
- We discuss how the limitations may be overcome to improve the automatic API-usage updates for Android apps.
- We release a replication package to allow easy replication and evaluation of our experiments and results.[2]

The remainder of this paper is structured as follows. Section II describes AppEvolve, the state-of-the-art work on automated updates of deprecated API usages in Android apps. Section III presents our replication settings. Section IV presents our findings. Section V presents our mitigation strategy and results for some failed updates of Android apps. Section VI discusses limitations and threats to validity of our findings. Section VII presents related work. Finally, Section VIII concludes and presents promising future work.

II. AppEvolve

We describe the problem statement, the approach, the dataset, and the main experimental results of AppEvolve.

A. Problem Statement

AppEvolve targets the case where a set of one or more API methods of the Android framework are deprecated and replaced by another set of one or more API methods in a new version of the framework, and where the Android apps that use the old APIs need to be updated. The goal of AppEvolve is to automatically update such Android apps so that they use the new APIs, while maintaining backward compatibility.

As an example, in Android API version 23, the method `getCurrentHour()`, that had been present in the Android API since version 1, was deprecated and the method `getHour()` was suggested as its replacement. Figure 1 shows an example that modifies, among other changes, a usage of the method `getCurrentHour()` to its replacement method `getHour()`. Given this example and the mapping `getCurrentHour` $\rightarrow$ `getHour`, AppEvolve learns a generic patch for `getCurrentHour()` as shown in Figure 2. The generic patch updates a deprecated method `getCurrentHour` usage to a replacement method `getHour` usage. *I* and *M* mean that the corresponding piece of code are inserted and modified, respectively. Backward compatibility is achieved by checking the version number of the mobile OS running the app. If the version is greater than or equal to a certain version number, the replacement method is used. Otherwise, the deprecated method is used.

Note that AppEvolve can only perform API usage learning and update one mapping at a time. Thus, even though `getCurrentMinute` is also a deprecated method, AppEvolve does not learn how to update it since this case is the mapping `getCurrentHour` $\rightarrow$ `getHour`.

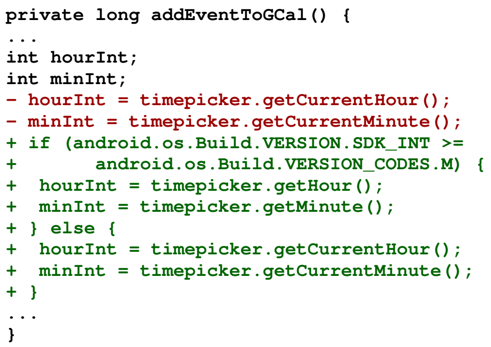
```
private long addEventToGCal() {
...
int hourInt;
int minInt;
- hourInt = timepicker.getCurrentHour();
- minInt = timepicker.getCurrentMinute();
+ if (android.os.Build.VERSION.SDK_INT >=
+       android.os.Build.VERSION_CODES.M) {
+  hourInt = timepicker.getHour();
+  minInt = timepicker.getMinute();
+ } else {
+  hourInt = timepicker.getCurrentHour();
+  minInt = timepicker.getCurrentMinute();
+ }
...
}
```

Fig. 1. A deprecated API-usage update

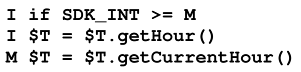
```
I if SDK_INT >= M
I $T = $T.getHour()
M $T = $T.getCurrentHour()
```

Fig. 2. AppEvolve's generic patch for updating uses of `getCurrentHour()`

B. Approach

AppEvolve takes as input a target app to update and a mapping from deprecated API method(s) to the corresponding replacement API method(s). Its processing is divided into four phases: *API-Usage Analysis*, *Update Example Search*, *Update Example Analysis* and *API-Usage Update*, as shown

[2]https://sites.google.com/smu.edu.sg/appevolve-replication

in Figure 3, a simplified version we recreated according to the Figure 1 in the AppEvolve paper [10].

In the *API-Usage Analysis* phase, AppEvolve accepts as input an *API Usage Change* and a *Target App*. An *API Usage Change* describes the mapping from the deprecated API method(s) to the corresponding replacement API method(s). Information about this API usage change may come, for example, from the documentation of the API itself. A *Target App* is an app that contains a usage of the deprecated API method and requires updates to make use of the replacement API method. Given these two inputs, AppEvolve pinpoints the location of the deprecated API usage in the target app. In the *Update Example Search* phase, AppEvolve searches GitHub for examples of API usage updates that modify the usage of the deprecated API method to include the usage of the replacement API methods. In the *Update Example Analysis* phase, AppEvolve generates generic patches from the examples of API usage updates and ranks these patches. In the *API Usage Update* phase, AppEvolve tries to apply the ranked patches one by one and returns the *Evolved Target App* if any of the edits is successful and validated.

We next describe each of the above phases in detail.

1) API-Usage Analysis: AppEvolve finds the location where the deprecated API method specified in the *API Usage Change* is used inside the *Target App*. Consider the deprecated API method `getCurrentHour()` mentioned in Section II-A. In this case, the *API Usage Change* is the replacement of `getCurrentHour()` with `getHour()`. Given this information, AppEvolve finds the location in the *Target App* that invokes `getCurrentHour()`.

2) Update Example Search: AppEvolve searches for apps in GitHub that use both the deprecated and the replacement API methods in their latest versions. For each such app, AppEvolve looks through the app history to find the commit where the uses of the replacement API methods are added. This is intended to find examples that produce a backward compatible Android app. For each of these apps, AppEvolve finds the changes that add the replacement API method to the app code that already contains the deprecated API method. These changes are the examples that can be used to update deprecated API method usages in other apps.

3) Update Example Analysis: Given the examples found in GitHub, AppEvolve translates each example to a generic patch. It does so by identifying edits related to the API usage via intraprocedural forward and backward dependency analysis on the variables involved in the API usage. Variables that are used in statements affected by the edits but not defined by the edits themselves are considered as context variables. All variables in the edits are then abstracted. Given the generic patches, AppEvolve computes the common core of these generic patches, defined as the longest subsequence of edits that are shared across the patches. The generic patches are then ranked based on their distance to the core.

4) API-Usage Update: Given a *Target App*, AppEvolve applies the generic patches according to their rank computed in the previous phase. When applying a patch, AppEvolve first finds mappings of the context variables to variables in the *Target App*. For each such mapping, AppEvolve then tries to apply the patch. If the edits are successfully applied, AppEvolve returns the *Evolved Target App*.

C. Experiments

For the target apps, the AppEvolve paper used 15 real-world apps from the F-droid repository.[3] These apps comprise five apps for each of Android API versions 22, 23, and 25, and cover 20 APIs used in 41 locations. For each API version, the API change is manually identified by reading the API documentation. Using the API change, the API usage in each app is guaranteed to be: (1) different from the ones in the other apps and (2) updated in the subsequent API version. When applied to the 15 apps considered, AppEvolve was able to update 17 out of 20 API usages (85% success rate) and 37 out of 41 (90% success rate) of their occurrences across the apps.

III. Replication Settings

The goal of our replication study is to determine whether the observed effectiveness of AppEvolve generalizes to a wider range of apps.

A. Dataset

Our replication study focuses on the same set of deprecated API methods as the original evaluation of AppEvolve and relies on the same set of change examples, but considers a different and larger set of mobile apps that use these deprecated methods. We find the mobile apps for our dataset by querying GitHub Code Search[4] using the names of the APIs. GitHub Code Search returns a list of ranked files matching the query. Since Github Code Search only supports textual queries, it returns many false positives, i.e., files that do not actually use the API methods that were queried for. Thus, we manually check that the considered files contain usages of the desired deprecated APIs. We additionally check that the considered files do not use the replacement APIs. From the files that pass these checks, we randomly select 54 API usages, each from a different app, as our dataset.

We note that although the AppEvolve change examples also come from GitHub, there is no overlap with our dataset. GitHub Code Search only indexes the latest version of each repository. The AppEvolve change examples consist of apps where the latest version uses the replacement API, while our dataset consists of apps that do not use the replacement API. Thus, no overlap is possible.

B. Procedure

For each app in our dataset, we must create an AppEvolve configuration. To do so, we first carefully studied the configurations that were used in the original AppEvolve experiments. We also asked the first author of AppEvolve to confirm how to configure AppEvolve correctly. Finally, we ran our

[3]https://fdroid.org
[4]https://github.com/search

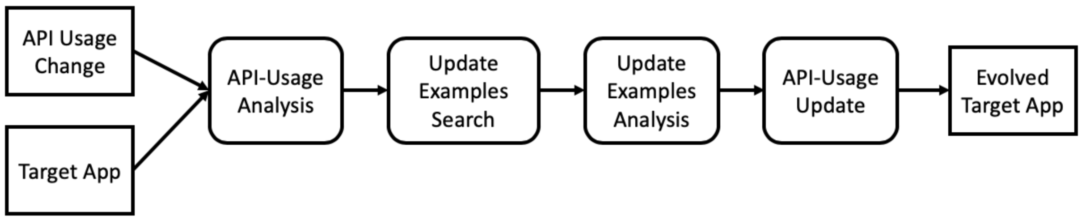

Fig. 3. Framework of AppEvolve

TABLE I
UPDATE FINDINGS STATISTICS (SOME OCCURRENCES OF DEPRECATED APIS ARE IN MULTIPLE CATEGORIES)

Category	Subcategory	Occurrences
Statements in the examples and at the target location are structurally different	Return statement	6
	If statement	1
	Method argument	4
	Arithmetic operand	3
	Declared variable	12
Object and arguments of the deprecated API method are in the form of complex expressions	-	20
Edits beyond method boundaries	-	3
Specific programming-language features	Inheritance	2
	Static modifier	1
	Final modifier	1
No examples	-	1
Others	-	4

experiments in the virtual machine environment provided by the AppEvolve authors.[5]

After configuring AppEvolve for the apps, we ran AppEvolve on them. We recorded the number of applicable and failed updates. We then categorized the failed updates using card sorting [28]. For this, we performed multiple passes on the failed updates. In the first pass, we put each of the failed updates into a category created based on our understanding of the reason for the failure. In the subsequent passes, we reevaluated the categories. We might rename a category to be more descriptive of the problem that occurs in the set of updates belonging to the category, or merge related categories into one. These steps were repeated until there were no more changes to the categories.

IV. FINDINGS

On our dataset, AppEvolve produces 10 applicable updates and 44 failed updates (19% success rate). The original AppEvolve evaluation also showed 4 failed updates. After investigating the 48 failed updates, we found some common reasons for the failures, as described below. Table I shows the number of occurrences of each of these issues.

1) The statements in the examples and at the target location are structurally different. AppEvolve infers edit operations at the statement level (insert a statement, move a statement, etc). Accordingly, AppEvolve is not able to apply changes inferred about method uses found in one kind of statement to a method use found in another kind of statement. We have observed that in the examples used in the evaluation of AppEvolve, it often occurs that the invocation of the deprecated API method is used as the right hand side of an assignment, while in our dataset the API method invocations occur in a variety of expression contexts. We now present some examples:

- Return statement (left side of Listing 1 in Table II): An invocation of the deprecated method `fromHtml` appears as part of a return statement of the `getTitle` method.
- If statement test (left side of Listing 2 in Table II): An invocation of the deprecated method `requestAudio-Focus` appears as a subexpression of the test expression of a conditional.
- Method argument (left side of Listing 3 in Table II): An invocation of the deprecated method `getCurrentHour` appears as the second argument of the invocation of method `String.format`.
- Binary operator (left side of Listing 4 in Table II): An invocation of the deprecated method `getCurrentHour` appears as the left argument of a concatenation with `":"`.
- Declared variable (left side of Listing 5 in Table II): An invocation of the deprecated method `getCurrentHour` appears as the initial value of a declared variable. Even though this case involves an assignment, such code does not match examples where the assignment involves a previously declared variable.

2) Object and arguments of the deprecated API method are complex expressions. In creating edits, AppEvolve only abstracts over variables. Accordingly, when examples always contain variables for the object or the arguments, the generated edits are not sufficient to update code in which these subterms are expressed as more complex expressions.

In the code shown on the left side of Listing 6 in Table II, an invocation of the deprecated method `setTextAppearance` involves expressions that do not have the form of a simple variable for both the object and arguments.

3) Edits beyond method boundaries. AppEvolve cannot learn edits that modify program elements that reside outside of the method containing the API usage to be updated. These edits include operations such as adding imports and adding fields to a class. Below is a snippet of an update example that involves edits beyond method boundaries.

[5]https://sites.google.com/view/appevolve

```
18a19
+ import android.location.GnssStatus;
22a24
+ import android.os.Build;
33a36,37
+ private GnssStatus.Callback callback;
41c45,51
- locationManager.addGpsStatusListener(listener);
+ if (android.os.Build.VERSION.SDK_INT >=
            android.os.Build.VERSION_CODES.N) {
+   locationManager.registerGnssStatusCallback(callback);
+ }
+ else{
+   locationManager.addGpsStatusListener(listener);
+ }
```

The above example involves a use of the deprecated API `addGpsStatusListener`. Updating a use of this API requires adding a private field `callback` of type *GnssStatus.Callback*. Since AppEvolve only learns edits inside a method containing the deprecated API usage, it misses this necessary addition. In the AppEvolve paper, this category is described as missing context information.

4) Specific programming language features. AppEvolve fails to update cases in which the update involves programming language features such as:

- Static modifier: The code on the left hand side of Listing 7 in Table III shows an example of this case. The deprecated API method `requestAudioFocus` is invoked by `mAudioManager`, which is a static field.
- Final modifier: The code in the left hand side of Listing 8 in Table III shows an example of this case. Variable `paint` that is used as the fourth argument to the deprecated API method `saveLayer` is a final field.
- Inheritance: The code below uses the deprecated API method `setAudioStreamType` of the class `MediaPlayer`. In the code snippet, `TestMediaPlayer` extends `MediaPlayer` and thus it inherits the `setAudioStreamType` method. This method should be updated but AppEvolve does not seem to recognize it due to the use of inheritance.

```
public class TestMediaPlayer extends MediaPlayer {
  public TestMediaPlayer() {
    setAudioStreamType(AudioManager.STREAM_MUSIC);
  }
  public TestMediaPlayer(Context testContext,
      int withResource) throws Exception {
    this();
    AssetFileDescriptor afd = testContext.getResources()
        .openRawResourceFd(withResource);
    assertNotNull(afd);
    setDataSource(afd.getFileDescriptor(),
      afd.getStartOffset(),afd.getLength());
    afd.close();
    prepare();
  }
}
```

5) No examples. No example illustrating the API update can be found in GitHub.

6) Others. These include cases that cannot be put to any category above.

The categories *edits beyond method boundaries* and *no examples* are from target apps in the original AppEvolve dataset and these categories are reported in the AppEvolve paper. The other categories are uncovered from our dataset.

V. Mitigations

To mitigate the observed failures, we tried to modify the deprecated API usages in the target app's source code so that AppEvolve could produce an applicable update. Looking at the 10 apps that were updated successfully, we observe that in each case the invocation of the deprecated method appears as the right-hand side of an assignment statement, or any arguments of the deprecated method are simple variables. We transform the remaining apps where possible accordingly, converting the invocations of the deprecated APIs to a form reminiscent of the three address code used in compiler intermediate representations [1].

In essence, our mitigations refactor the deprecated method invocation in the target app to resemble the examples from which AppEvolve learns edits. The right side of Table II shows a diff that applies these mitigations on the cases presented in Section IV where the mitigation allows AppEvolve to produce applicable updates.

1) After the refactoring described in Listing 1 Table II, rather than directly returning the result of invoking the deprecated method `fromHtml`, we first assign it to a fresh variable named `a` of type `Spanned`.
2) After the refactoring described in Listing 2 of Table II, rather than directly inserting the constants `AudioManager.STREAM_MUSIC` and `AudioManager.AUDIOFOCUS_GAIN` as the second and third arguments of `requestAudioFocus`, we first assign them to fresh `int`-typed variables `arg1` and `arg2`, respectively. Moreover, the result of invoking `requestAudioFocus` is assigned to a fresh variable named `res` of type `AudioManager` that is then used in the if condition.
3) After the refactoring described in Listing 3 of Table II, rather than directly inserting the result of invoking `getCurrentHour` as the second argument of `String.format`, the result is first assigned to fresh a variable named `hour` of type `int`.
4) After the refactoring described in Listing 4 of Table II, rather than concatenating the result of `getCurrentHour` directly to ":", the result is first assigned to a fresh variable named `hour` of type `int`.
5) After the refactoring described in Listing 5 of Table II, rather than directly assigning the result of invoking `getCurrentHour` to the variable `hours` when it is declared, we declare `hours` first and then assign the result of invoking `getCurrentHour` to `hours`.
6) After the refactoring described in Listing 6 of Table II, rather than invoking `setTextAppearance` directly from the object returned by invoking `findViewById`, the returned object is first assigned to a variable named `t` of type `TextView`. `setTextAppearance` is then invoked from `t`.

Our mitigations result in 38 successful updates out of the 44 failed updates on the apps in our dataset (86% success rate),

TABLE II
SUCCESSFUL SIMPLE REFACTORING MITIGATIONS THAT ALLOW APPEVOLVE TO GENERATE APPLICABLE UPDATES (PART I)

Listing	Original Code	Refactoring Diff

1. Structurally different: return statement	Spanned getTitle(){
	    return Html.fromHtml(title);
	}
		Spanned getTitle(){
		+ Spanned a;
		+ a = Html.fromHtml(title);
		− return Html.fromHtml(title);
		+ return a;
		}

2. Structurally different: if statement	public void play() {
	    if (mAudioManager.requestAudioFocus(
	            mAudioFocusListener,
	            AudioManager.STREAM_MUSIC,
	            AudioManager.AUDIOFOCUS_GAIN) !=
	        AudioManager.AUDIOFOCUS_REQUEST_GRANTED){
	        return;
	    }
	}
		public void play() {
		− if (mAudioManager.requestAudioFocus(
		− mAudioFocusListener,
		− AudioManager.STREAM_MUSIC,
		− AudioManager.AUDIOFOCUS_GAIN) !=
		− AudioManager.AUDIOFOCUS_REQUEST_GRANTED){
		− return;
		− }
		+ int res;
		+ int arg1=AudioManager.STREAM_MUSIC;
		+ int arg2=AudioManager.AUDIOFOCUS_GAIN;
		+ res = mAudioManager.requestAudioFocus (
		    mAudioFocusListener, arg1, arg2);
		+ if (res != AudioManager
		+ .AUDIOFOCUS_REQUEST_GRANTED) {
		+ return;
		+ }
		}

3. Structurally different: method argument	protected String getInputDataString() {
	    return String.format("\%02d:\%02d",
	    timePicker.getCurrentHour(),
	    timePicker.getCurrentMinute());
	}
		protected String getInputDataString() {
		+ int hour;
		+ hour = timePicker.getCurrentHour();
		+ return String.format("%02d:%02d", hour,
		+ timePicker.getCurrentMInute());
		− return String.format("\%02d:\%02d",
		− timePicker.getCurrentHour(),
		− timePicker.getCurrentMinute());
		}

4. Structurally different: arithmetic operand	public void displayTime(View view) {
	    String time = timePicker.getCurrentHour()
	       + ":" + timePicker.getCurrentMinute();
	    Toast.makeText(this, time,
	        Toast.LENGTH_SHORT).show();
	}
		public void displayTime(View view) {
		+ int hour;
		+ hour = timePicker.getCurrentHour();
		+ String time = hour + ":" +
		+ timePicker.getCurrentMinute();
		− String time = timePicker.getCurrentHour()
		− + ":" + timePicker.getCurrentMinute();
		    Toast.makeText(this, time,
		        Toast.LENGTH_SHORT).show();
		}

5. Direct assignment after declaration	public Schedule generate() {
	    TimePicker timePicker = (TimePicker) activity
	        .findViewById(R.id.timePicker);
	    int hours = timePicker.getCurrentHour();
	    int minutes = timePicker.getCurrentMinute();
	    SeekBar seekBar = (SeekBar) activity
	        .findViewById(R.id.setLuminosity);
	    int luminosity = seekBar.getProgress();
	    return new Schedule(hours, minutes,
	                    luminosity);
	}
		public Schedule generate() {
		    TimePicker timePicker = (TimePicker) activity
		        .findViewById(R.id.timePicker);
		+ int hours;
		+ hours = timePicker.getCurrentHour();
		− int hours = timePicker.getCurrentHour();
		    int minutes = timePicker.getCurrentMinute();
		    SeekBar seekBar = (SeekBar) activity
		        .findViewById(R.id.setLuminosity);
		    int luminosity = seekBar.getProgress();
		    return new Schedule(hours, minutes,
		                    luminosity);
		}

6. Complex expressions	protected void onClick() {
	    ...
	    Dialog d = builder.create();
	    d.show();
	    ((TextView) d.findViewById(android.R.id.
	      message))
	        .setTextAppearance(getContext(),
	        android.R.style.TextAppearance_Small);
	    ...
	}
		protected void onClick() {
		    ...
		    Dialog d = builder.create();
		    d.show();
		− ((TextView) d.findViewById(android.R.id.
		− message))
		− .setTextAppearance(getContext(),
		   − android.R.style.TextAppearance_Small);
		+ Context c = getContext();
		+ int i = android.R.style.
		+ TextAppearance_Small;
		+ TextView t = ((TextView) d.findViewById
		+ (android.R.id.message));
		+ t.setTextAppearance(c, i);
		    ...
		}

TABLE III
SUCCESSFUL SIMPLE REFACTORING MITIGATIONS THAT ALLOW APPEVOLVE TO GENERATE APPLICABLE UPDATES (PART II)

Listing	Original Code	Refactoring Diff

7. Incomplete support: static	`private static AudioManager mAudioManager;`
	`private static OnAudioFocusChangeListener`
	`     afChangeListener;`
	`public static void requestAudioFocus`
	`  (Context context) {`
	`  if(mAudioManager == null){`
	`    mAudioManager = (AudioManager)`
	`      context.getApplicationContext()`
	`      .getSystemService`
	`      (Context.AUDIO_SERVICE);`
	`  }`
	`  mAudioManager.requestAudioFocus(`
	`    afChangeListener,`
	`    AudioManager.STREAM_MUSIC,`
	`    AudioManager.AUDIOFOCUS_GAIN);`
	`}`
		`private static AudioManager mAudioManager;`
		`private static OnAudioFocusChangeListener`
		`     afChangeListener;`
		`public static void requestAudioFocus`
		`  (Context context) {`
		`  if(mAudioManager == null){`
		`    mAudioManager = (AudioManager)`
		`      context.getApplicationContext()`
		`      .getSystemService`
		`      (Context.AUDIO_SERVICE);`
		`  }`
		`- mAudioManager.requestAudioFocus(`
		`-   afChangeListener,`
		`-   AudioManager.STREAM_MUSIC,`
		`-   AudioManager.AUDIOFOCUS_GAIN);`
		`+ int res;`
		`+ int a = AudioManager.STREAM_MUSIC;`
		`+ int b = AudioManager.AUDIOFOCUS_GAIN;`
		`+ AudioManager am = mAudioManager;`
		`+ AudioManager.OnAudioFocusChangeListener`
		`+      af = afChangeListener;`
		`+ res = am.requestAudioFocus(af,a,b);`
		`}`

8. Incomplete support: final	`private final Paint paint;`
	`@Override protected void onDraw(Canvas canvas) {`
	`  super.onDraw(canvas);`
	`  canvas.drawColor(Color.GREEN);`
	`  canvas.saveLayer(0, 0, canvas.getWidth(),`
	`    canvas.getHeight(), paint,`
	`    Canvas.ALL_SAVE_FLAG);`
	`  canvas.restore();`
	`}`
		`private final Paint paint;`
		`@Override protected void onDraw(Canvas canvas) {`
		`  super.onDraw(canvas);`
		`  canvas.drawColor(Color.GREEN);`
		`+ float a = 0;`
		`+ float b = 0;`
		`+ float c = getWidth();`
		`+ float d = getHeight();`
		`+ int flag = Canvas.CLIP_SAVE_FLAG;`
		`+ Paint p;`
		`+ p = paint;`
		`+ canvas.saveLayer(a, b, c, d, p, flag);`
		`- canvas.saveLayer(0, 0, canvas.getWidth(),`
		`-   canvas.getHeight(), paint,`
		`-   Canvas.ALL_SAVE_FLAG);`
		`  canvas.restore();`
		`}`

overall giving success on 48 out of the 54 apps in our dataset (89% success rate).

VI. DISCUSSION

In this section, we discuss the relation of our results to those originally presented for AppEvolve, future directions for improving the approach to updating deprecated API usage in Android apps, and threats to validity.

A. *Result Assessment*

We were initially surprised by the large difference between the results on our data set, where we obtained a success rate of 19%, and the results reported by the authors of AppEvolve, who obtained a success rate of 85%, even though our dataset was constructed randomly, without taking into account the strengths or weaknesses of AppEvolve. Indeed, given that AppEvolve relies on matching statements, abstracting only over variables, it is surprising that AppEvolve could achieve such high accuracy on e.g. simple getter methods such as `getCurrentHour()` (e.g., Listing 3). In our dataset, such methods are often used as subexpressions of arbitrarily complex statements, which may contain project-specific code. Likewise, as illustrated by Listing 2, invocations of method calls may have arbitrarily complex, possibly project-specific, arguments that are not likely to be found in other codebases.

To understand better how AppEvolve is able to achieve a high rate of success in the previously reported evaluation, we investigate the examples and target apps used. Given the difficulty of finding the original files for the examples, for which no origin information is provided, we focus on a single example, the call to `getCurrentHour()` in the app CONVERSATIONS version 1.8.0 in F-Droid (A02-U03 in the AppEvolve paper). In F-Droid, we find that the only invocation of `getCurrentHour()` occurs in the argument list of a call to the `set` method of the `Calendar` class, a code structure similar to the code shown on the left side of Listing 3. In the AppEvolve dataset, this call is extracted into a separate statement, analogous to our mitigation shown on the right side of Listing 3. We have furthermore looked at one of the examples provided for this API, and have found the original code on GitHub in the file `RepeaterDialogFragment.java` from the `marekpiotrowskimp/nyndro_remote` repository. Again, we find that in the GitHub code, the call appears in an argument list, while the corresponding example in the AppEvolve example set has been transformed analogous to our mitigation. These changes convert invocations that occur in differing contexts to invocations that can be exploited by AppEvolve.

We note that the observed issue appears to be a direct result of the key design decision of AppEvolve, of not merging examples. Merging could motivate discarding irrelevant context of the use of the deprecated method and extending the abstraction of subterms beyond variables, but, as observed by the authors of AppEvolve, it could discard some information that is necessary to correctly perform the transformation.

B. Future Directions

Code normalization. Our above analysis suggests that our mitigation succeeds because the AppEvolve authors have already performed such a mitigation in the change examples included in the AppEvolve replication package. To improve the results of AppEvolve in the more general case, a solution could be to systematically first normalize the change examples to a standard form. Edits can then be learned from this standard form. When the edit is applied to a new piece of code, that target code should also be normalized, to minimize the code variations. If the edits are successfully applied, the resulting code can then be refactored again to restore the coding style of the developers. Our mitigations represent a subset of the normalizations that may be possible.

Coding Style and Readability. Any tool that automatically transforms code runs the risk of converting the code to a style that is incompatible with the preferences of the given app's developers. In the case of AppEvolve, there is already the risk of changing the existing coding style to the coding style found in the app providing the change example, and any normalization performed to the target code further increases this risk. To assess the impact of AppEvolve on coding style, we perform a simple experiment to compare the output of AppEvolve after applying our mitigation with code that is manually updated to deal with a set of deprecated APIs. We asked a software engineer who is not an author of this paper to update the code shown on the left hand side of Listing 4 in Table II. The code generated by AppEvolve and by the engineer is shown below.

AppEvolve's Update

```
public void displayTime(View view){
  int hour;
  if (android.os.Build.VERSION.SDK_INT >=
      android.os.Build.VERSION_CODES.M) {
    hour=timePicker.getHour();
  }
  else {
    hour=timePicker.getCurrentHour();
  }
  String time=hour + ":" + timePicker.getCurrentMinute();;
  Toast.makeText(this,time,Toast.LENGTH_SHORT).show();
}
```

Engineer's Update

```
public void displayTime(View view) {
    String time;
    if (android.os.Build.VERSION.SDK_INT >= 23) {
        // only for marshmallow and newer versions
        time = timePicker.getHour() + ":" +
                timePicker.getCurrentMinute();
    } else {
        time = timePicker.getCurrentHour() + ":" +
                timePicker.getCurrentMinute();
    }
    Toast.makeText(this, time, Toast.LENGTH_SHORT).show();
}
```

Semantic Preservation. The code made manually by the engineer is substantially different from the code produced by AppEvolve, and developers may indeed prefer the code made by the engineer. In particular, the engineer did not introduce fresh variables and provide a more concise code. Future work may investigate how to restore the original coding style after applying AppEvolve. One step would be to choose good names for the newly introduced variables. Some recent studies use deep learning and mining software repositories to recommend method and class names [3]. These approaches can potentially be extended for inferring good variable names in our context.

C. Threats to Validity

One threat is related to whether we have configured AppEvolve correctly. To mitigate this threat, we have tried our best to run AppEvolve following the instructions given by the authors in the AppEvolve documentation. We have also asked AppEvolve's first author how to configure AppEvolve correctly for new mobile apps. Based on the information obtained, we independently reconstructed the configurations for the target apps in the original dataset, and using these configurations were able to reproduce the results reported for that dataset. We have also rechecked our configurations for the new dataset several times. We have released a replication package for others to check and validate.[6]

Another threat is related to the generalizability of the findings. We have added 54 new applications to evaluate the effectiveness of AppEvolve in generating applicable updates. This translates to 54 API usage locations on top of the 41 API usage locations in AppEvolve dataset, thus more than doubling the number of API usage locations and more than tripling the number of apps as compared to the dataset used in the original work. We believe this is sufficient to understand the capabilities of AppEvolve, as 44 of the 54 API usages that we have added uncover limitations of AppEvolve. There might be other cases that AppEvolve cannot handle, but we believe that we have found many of them.

VII. Related Work

A. API Deprecation

Many works have studied API deprecation. Zhou and Walker [30] found that, in practice, deprecating APIs does not always follow *deprecated-replace-remove* cycle, such as many deprecated APIs are undeprecated. They also developed a tool to warn about deprecated API usages in StackOverflow posts. Kapur et al. [14] found that APIs might be removed without being marked as deprecated. Raemaekers et al. [21] discovered that some Java artifacts on the Maven Central Repository never remove deprecated APIs. Brito et al. [7] showed that not all

[6]https://sites.google.com/smu.edu.sg/appevolve-replication

APIs are annotated with replacement messages. Robbes et al. [22] analyzed the Smalltalk ecosystem and showed that some API changes caused by deprecation can substantially impact the ecosystem. This study was replicated on the Java ecosystem and similar results were reported [26], [27], except that the number of deprecated API replacements was higher in the Smalltalk ecosystem. Sawant et al. [25] created a taxonomy containing 12 reasons for deprecation and developed an approach to automatically classify them. Li et al. [16] performed an exploratory study on characterizing Android APIs. They found that, among other things, deprecated Android APIs are not always consistently annotated and documented, and they are also regularly cleaned up.

All of the above studies aim to understand API deprecation. In this work, we aim to understand the applicability of a state-of-the-art approach for automatic update of Android apps, which updates usages of deprecated methods to corresponding usages of their replacement methods.

B. Replication Studies

There have been a number of replication studies in the software engineering domain that have provided new insights about the replicated studies. Chen and Jiang [8] replicated a study by Yuan et al. [29] of logging practices. In contrast to the observations of Yuan et al., Chen and Jiang found that bug reports without a log message take a shorter time to resolve than bug reports that include a log. Greiler et al. [11] replicated the work of Bird et al. [6] on the correlation between code ownership and software quality. Greiler et al. used new and refined code ownership metrics and prediction models. Akbarinasaji et al. [2] replicated and reinforced the finding on the bug fixing time estimation model by showing similar result with the previous work. A replication study on open source development by Trong et al. [9] found new findings from the previous work by Mockus et al. [18]. They supported some of the previous hypotheses and proposed revisions on hypothesis related to the need of formal arrangement for work coordination and on hypothesis regarding the number of core developers on open source project. A replication study by Orru et al. [20] conducted an analysis of the use of inheritance in Python systems that was previously done on Java. Their result shows that compared on the previous findings on Java, Python has more classes that are inherited from but fewer classes that inherit from other classes. Our replication study highlights unreported limitations of the studied tool.

C. Program Transformation

Normalization of code to improve the results of program transformations has a long history. It was extensively used in the early 1990s in the context of partial evaluation (specialization of programs to the values of some known inputs), in the form of *binding-time improvements*, in order to bring known values closer together, to allow code simplifications [13]. Namjoshi and Pavlinovic have more generally studied and formalized the impact of program transformations on program analysis [19], considering both positive and negative impacts on precision.

AppEvolve is part of a line of research that has recently been attracting increasing interest on inferring program transformations from one or more examples. Some that could potentially benefit from our mitigation include LASE [17] and REFAZER [23]. LASE [17] creates an edit script from common changes in a set of examples, locates the edit locations in a target application, and does the code transformation automatically. However LASE is only able to abstract variable, method and type names, and thus it will never consider a method invocation with a simple variable as an argument to be the same as a method invocation where the argument is a more complex expression. Thus, it could potentially benefit from our mitigation. REFAZER [23] learns rewrite rules from change examples provided by the user. Rewrite rules use some information about the change context to determine whether a rule should be applied, which can limit rule applicability when a deprecated API is used in different contexts. Applying our mitigation to the context could reduce the diversity and increase the rule applicability. Some other transformation-rule inference systems, such as PHOENIX [5], Spdiff [4], and REVISAR [24] can describe terms appearing in arbitrary contexts and containing arbitrary subexpressions, and may thus benefit less or not at all from our mitigation.

VIII. Conclusion and Future Work

AppEvolve is the state-of-the-art approach for automatic update of deprecated-API usage in Android apps. Experiments previously reported for AppEvolve have shown that it can generate applicable updates for 85% of these API changes and 90% of their usage locations on mobile apps in the AppEvolve dataset.

In this work, we evaluate whether this observed effectiveness is generalizable. We add 54 additional mobile apps that use the APIs contained in the AppEvolve dataset. Running AppEvolve on these mobile apps shows that AppEvolve fails to generate applicable updates for 81% of the mobile apps. By analyzing these failed cases, we found that they failed mainly due to:

1) Statements in the examples and at the target location are structurally different.
2) Object and arguments of the deprecated API method are in the form of complex expressions.
3) Edits are required beyond method boundaries.
4) Specific programming language features.
5) No examples found in GitHub.

We mitigate the first and the second categories by performing a simple refactoring that modifies the code containing API usage in the target app to resemble the one in the example. Our mitigations enable AppEvolve to generate applicable updates for 86% of the failed cases.

To try to resolve the discrepancy in the findings, we looked closer at the examples and target apps used by finding the original examples and target apps. We found that they were transformed analogous to our mitigation. These transformations convert invocations in various contexts into code exploitable by AppEvolve.

Based on our findings, we propose future directions for automatic update of deprecated-API usage in Android apps, which include code normalization to ensure that both the example of API usage and the target app code are in the same standard form to minimize the variety of ways that the code is written. Using this technique, simple refactoring that we use to mitigate the failed cases may not be necessary. We also propose the usage of identifier name recommendation to name new variables that may have been introduced due to code normalization.

Acknowledgement

This research was supported by the Singapore National Research Foundation (award number: NRF2016-NRF-ANR003).

References

[1] Alfred V. Aho, Sethi Ravi, and Jeffrey D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley Pub. Co., Reading, Mass., 1986.

[2] Shirin Akbarinasaji, Bora Caglayan, and Ayse Bener. Predicting bug-fixing time. *J. Syst. Softw.*, 136(C):173–186, February 2018.

[3] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49. ACM, 2015.

[4] Jesper Andersen, Anh Cuong Nguyen, David Lo, Julia L. Lawall, and Siau-Cheng Khoo. Semantic patch inference. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 382–385, New York, NY, USA, 2012. ACM.

[5] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *ESEC/FSE*, pages 613–624, New York, NY, USA, 2019. ACM.

[6] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: Examining the effects of ownership on software quality. In *ESEC/FSE*, pages 4–14, New York, NY, USA, 2011. ACM.

[7] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. Do developers deprecate APIs with replacement messages? a large-scale analysis on Java systems. In *SANER*, volume 1, pages 360–369. IEEE, 2016.

[8] Boyuan Chen and Zhen Ming (Jack) Jiang. Characterizing logging practices in Java-based open source software projects — a replication study in apache software foundation. *Empirical Softw. Engg.*, 22(1):330–374, February 2017.

[9] Trung T. Dinh-Trong and James M. Bieman. The freebsd project: A replication case study of open source development. *IEEE Trans. Softw. Eng.*, 31(6):481–494, June 2005.

[10] Mattia Fazzini, Qi Xin, and Alessandro Orso. Automated API-usage update for Android apps. In *ISSTA*, pages 204–215. ACM, 2019.

[11] Michaela Greiler, Kim Herzig, and Jacek Czerwonka. Code ownership and software quality: A replication study. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 2–12. IEEE Press, 2015.

[12] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. Understanding and detecting evolution-induced compatibility issues in Android apps. In *ASE*, pages 167–177. ACM, 2018.

[13] N.D. Jones, C.K. Gomard, , and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[14] Puneet Kapur, Brad Cossette, and Robert J. Walker. Refactoring references for library migration. In *OOPSLA*, pages 726–738. ACM, 2010.

[15] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. Cid: Automating the detection of API-related compatibility issues in Android apps. In *ISSTA*, pages 153–163. ACM, 2018.

[16] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated Android APIs. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, pages 254–264. ACM, 2018.

[17] Na Meng, Miryung Kim, and Kathryn S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *ICSE*, pages 502–511. IEEE Press, 2013.

[18] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, July 2002.

[19] Kedar S. Namjoshi and Zvonimir Pavlinovic. The impact of program transformations on static program analysis. In *Static Analysis - 25th International Symposium (SAS)*, volume 11002 of *Lecture Notes in Computer Science*, pages 306–325. Springer, 2018.

[20] Matteo Orrù, Ewan Tempero, Michele Marchesi, and Roberto Tonelli. How do Python programs use inheritance? a replication study. In *Asia-Pacific Software Engineering Conference (APSEC)*, December 2015.

[21] Steven Raemaekers, Arie Van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 215–224. IEEE, 2014.

[22] Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to API deprecation?: the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*, page 56. ACM, 2012.

[23] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *ICSE*, pages 404–415. IEEE Press, 2017.

[24] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, and Loris D'Antoni. Learning quick fixes from code repositories. *CoRR*, abs/1803.03806, 2018.

[25] Anand Ashok Sawant, Guangzhe Huang, Gabriel Vilen, Stefan Stojkovski, and Alberto Bacchelli. Why are features deprecated? an investigation into the motivation behind deprecation. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 13–24. IEEE, 2018.

[26] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of 25,357 clients of 4+ 1 popular Java APIs. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 400–410. IEEE, 2016.

[27] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of clients of 4+ 1 popular Java APIs and the JDK. *Empirical Software Engineering (EMSE)*, 23(4):2158–2197, 2018.

[28] D. Spencer. *Card Sorting: Designing Usable Categories*. Rosenfeld Media, 2009.

[29] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *ICSE*, pages 102–112. IEEE Press, 2012.

[30] Jing Zhou and Robert J Walker. API deprecation: a retrospective analysis and detection method for code examples on the web. In *ICSE*, pages 266–277. ACM, 2016.